Evolutionary Algorithms for Finding Short Addition Chains: Going the Distance

Stjepan Picek[1], Carlos A. Coello Coello[2], Domagoj Jakobovic[3], Nele Mentens[1]

[1] KU Leuven, ESAT/COSIC and iMinds, Kasteelpark Arenberg 10, bus 2452, B-3001 Leuven-Heverlee, Belgium
[2] CINVESTAV-IPN, Department of Computer Science, Av. IPN No. 2508, Col. San Pedro Zacatenco, Mexico, D.F. 07360, MEXICO
[3] University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia

Abstract. The problem of finding the shortest addition chain for a given exponent is of great relevance in cryptography, but is also very difficult to solve since it is an **NP**-hard problem. In this paper, we propose a genetic algorithm with a novel representation of solutions and new crossover and mutation operators to minimize the length of the addition chains corresponding to a given exponent. We also develop a repair strategy that significantly enhances the performance of our approach. The results are compared with respect to those generated by other metaheuristics for instances of moderate size, but we also investigate values up to $2^{127} - 3$. For those instances, we were unable to find any results produced by other metaheuristics for comparison, and three additional strategies were adopted in this case to serve as benchmarks. Our results indicate that the proposed approach is a very promising alternative to deal with this problem.

Keywords: Addition chains, Cryptography, Genetic Algorithms, Exponentiation

1 Introduction

Field or modular exponentiation has several important applications in error-correcting codes and cryptography. Well-known public-key cryptosystems such as Rivest-Shamir-Adleman (RSA) [1] adopt modular exponentiation. In a simplified way, modular exponentiation can be defined as the problem of finding the (unique) integer $B \in [1, \ldots, p-1]$ that satisfies:

$$B = A^c \bmod p, \tag{1}$$

where A is an integer in the range $[1, \ldots, p-1]$, c is an arbitrary positive integer and p is a large prime number. One possible way of reducing the computational load of Eq. (1) is to minimize the total number of multiplications required to compute the exponentiation.

Since the exponent in Eq. (1) is additive, the problem of computing powers of the base element A can be formulated as an addition calculation, for which so-called *addition chains* are used. Informally, an addition chain for the exponent c of length l is a sequence V of positive integers $v_0 = 1, \ldots, v_l = c$, such that for each $i > 1$, $v_i = v_j + v_k$ for some j and k with $0 \leq j \leq k < i$. An addition chain provides the correct sequence of multiplications required for performing an exponentiation. Thus, given an addition chain V that computes the exponent c as indicated before, we can find $B = A^c$ by successively computing: $A, A^{v_1}, \ldots, A^{v_{l-1}}, A^c$. For example, if we want to compute A^{60}, the traditional procedure would require 60 multiplications. However, if we use instead the following addition chain: $[1 \to 2 \to 4 \to 6 \to 12 \to 24 \to 30 \to 60]$, then only seven multiplications are required:

$$A^1; A^2 = A^1A^1; A^4 = A^2A^2; A^6 = A^4A^2; A^{12} = A^6A^6;$$
$$A^{24} = A^{12}A^{12}; A^{30} = A^{24}A^6; A^{60} = A^{30}A^{30}. \quad (2)$$

Thus, the length of the addition chain defines the number of multiplications required for computing the exponentiation. The aim is to find the shortest addition chain for a given exponent c (several addition chains can be produced for the same exponent). Naturally, as the exponent value grows, it becomes more difficult to find a chain that forms the exponent in a minimal number of steps.

One simple algorithm that can be used (although, in general it will not give optimal results) works in the following way. First, write the exponent in its binary representation. Then, replace each occurrence of the digit 1 with the letters "DA" and each occurrence of the digit 0 with the letter "D". After all digits are replaced, cross out the first "DA" that appears on the left. What remains represents a rule to calculate the exponent, since the letter "A" stands for addition (multiplication) and the letter "D" for doubling (squaring). If we consider again the example A^{60}, the exponent in binary representation would be "111100". After the replacement and the removal of "DA" at the left we have "DADADADD". Thus, the rule is: square, multiply, square, multiply, square, multiply, square, square ($1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 15 \to 30 \to 60$). This is a simple example describing *the binary method*. We can immediately observe that the binary method does not always give the shortest chain (cf. with the chain given in Eq. (2)). In fact, already for the value 15, the binary method will not produce the shortest chain [2]. However, the binary method can be generalized to some more powerful methods as presented in Section 2. Unfortunately, in general, the problem of finding the shortest addition chain is **NP**-hard [3]. This has motivated the use of metaheuristics to tackle this problem as indicated in Section 3.

Here, we propose a genetic algorithm to find short addition chains for a given exponent. Our main contributions are the following: the first one is a new representation of solutions. With that representation, we can obtain a better granularity than when using just the representation based on the values in the addition chains. Next, we present several mutation and crossover operators de-

signed to improve convergence. The behavior of those operators is modeled on the basis of several relevant test case scenarios as presented in Section 2. We then design *repair heuristics* that we believe are an integral part of the algorithm and we use several examples to justify our approach. From a more pragmatic perspective, in Section 5, we investigate a number of exponents that we want to obtain, whose values progress gradually from small ones up to the ones that are relevant in real-world applications. Finally, we identify a possible oversight in most of the relevant works that limits the applicability of those algorithms.

2 On Addition Chains

We start this section with basic notions about addition chains and, afterwards, we give several important results we use when designing our algorithm. Next, we briefly discuss algorithms that are commonly used to compute exponentiations. We follow the notation and theoretical results presented in "The Art of Computer Programming, Volume 2: Seminumerical Algorithms" [2]. For more detailed information about addition chains, we refer the readers to Chapter 4.6.3. "Evaluation of Powers" [2].

2.1 Theoretical Background

Definition 1 *An addition chain is a sequence of positive values starting with the value 1 and finishing with the desired exponent value n.*

Definition 2 *An addition chain is called ascending if:*

$$1 = a_0 < a_1 < a_2 < ... < a_r = n. \tag{3}$$

In this work, we focus only on ascending chains. From this point on, when we talk about addition chains, we mean ascending addition chains.

The values in the addition chain have the property that they are the sum of two values appearing previously in the chain. Formally, an addition chain is a sequence $a_0 = 1, a_1, ..., a_r = n$ where:

$$a_i = a_j + a_k, \text{ for some } k \leq j < i. \tag{4}$$

The shortest length of any valid addition chain is denoted as $l(n)$. In the length of a chain, one does not count the initial step that has a value of one.

Next, it is possible to define types of steps in the addition chain based on Eq. (4):

1. *Doubling step*; when $j = k = i - 1$. This step always gives the maximal possible value at the position i.
2. *Star step*; when j but not necessarily k equals $i - 1$.
3. *Small step*; when $log_2(a_i) = log_2(a_{i-1})$.
4. *Standard step*; when $a_i = a_j + a_k$ where $i > j > k$.

On the basis of the aforementioned steps, it is easy to infer the following conclusions: [2]:

- The first step is always a doubling step.
- A doubling step is always a star step and never a small step.
- A doubling step must be followed by a star step.

Now, we focus on the shortest addition chains. Trivially, the shortest chain for any number n must have at least $log_2(n)$ steps. To be more precise, any chain length is equal to $log_2(n)$ plus the number of small steps [2].

Let $\nu(n)$ be the number of ones in the binary representation of the exponent n. When $\nu(n) \geq 9$ then there are at least four small steps in any chain for exponent length n [4]. That statement can be also generalized with the following theorem [4]:

Theorem 1 *If $\nu(n) \geq 2^{4 \cdot m - 1} + 1$, then $l(n) \geq log_2(n) + m + 3$ where m is a nonnegative value.*

Definition 3 *A star chain is a chain that involves only star operations.*

The minimal length of a star chain is denoted as $l^*(n)$ and it holds:

$$l(n) \leq l^*(n). \tag{5}$$

Although it seems intuitive that the shortest addition chain is also a star chain, in 1958, Walter Hansen proved that for certain large exponents n, the value of $l(n)$ is smaller than $l^*(n)$ [2]. The smallest such exponent n equals 12 509.

Albeit counterintuitive, there exist values of n for which $l(n) = l(2n)$ with the smallest example being $n = 191$. Here, both n and $2n$ have length l equal to 11. Furthermore, there exist values of n where $l(n) > l(2n)$ [5]. The smallest such n is 375 494 703 [6].

Finally, the length seems to be the most difficult to compute for one specific class of numbers: let $c(r)$ be the smallest value of n such that $l(n) = r$ [2]. Therefore, $c(r)$ is the first integer value requiring r steps in a shortest addition chain [5]. To obtain such shortest addition chains is regarded more difficult than to obtain a shortest addition chain for some other value (of course, with regards to the size).

Up to now, we discussed only ascending addition chains, but there exists a number of other types of chains, e.g. addition-subtraction chains [2], differential addition chains [7] or differential addition-subtraction chains [7].

2.2 Techniques for Exponentiation

A number of techniques that are useful for cryptography, and that apply to both exponentiation in a multiplicative group and elliptic curve point multiplication, are explained in [3] and [8] and can be divided into three categories:

1. techniques for general exponentiation,

2. techniques for fixed-base exponentiation and
3. techniques for fixed-exponent exponentiation.

In the following paragraphs, we use the term exponentiation, while all principles hold for both exponentiation and elliptic curve point multiplication. In the first category, the most straightforward ways to perform an exponentiation or a point multiplication, are the left-to-right and right-to-left binary methods. An option for speeding up these algorithms consists of evaluating more than one bit of the exponent at a time after precomputing a number of multiples of the base. An example is the window or k-ary method that evaluates k bits of the exponent at a time. The precomputation of base multiples maximizes the speed by minimizing the number of multiplications. However, the optimizations require a larger memory usage for the storage of the precomputed values. When the base is fixed, the precomputed multiples of the base can be prestored, which shortens the time needed for the online exponentiation.

Another way of minimizing the number of multiplications without storing precomputed multiples of the base is exponent recoding, which uses a representation of the exponent that is different from the binary representation. The recoding of the exponent requires additional resources on a chip (logic gates) or a microprocessor (program memory).

For elliptic curve cryptography, further speed optimizations are possible by considering elliptic curves with special properties, like the Gallant-Lambert-Vanstone (GLV) curve [9], the Galbraith-Lin-Scott (GLS) curve [10] or the FourQ curve [11]. In [12], side-channel security is taken into account in the derivation of efficient algorithms for scalar multiplication on GLS-GLV curves.

In this paper, we focus on addition chains for fixed-exponent exponentiations or fixed-scalar point multiplication without taking into account optimizations using specific fields or curves. We do not consider side-channel analysis, but we believe this does not undermine our results, since a number of side-channel countermeasures can be applied on top of the proposed addition chains. Examples are point blinding or randomized projective coordinates [13].

3 Related Work

In 1990, Bos and Coster present the Makesequence algorithm that produces an addition sequence of a set of numbers [14]. The proposed method is able to find chains of large dimensions, and the authors conclude that their method is relatively more effective than the binary method. The heuristics in the algorithm choose, on the basis of a weight function, which method will be used to produce the sequence (the authors experiment with four methods). However, the authors report that their current weight function does not give satisfactory results and they experiment with simulated annealing, but without success.

Nedjah and de Macedo Mourelle experiment with a genetic algorithm (GA) in order to find minimal addition chains [15]. They use binary encoding where value 1 means that the entry number is in the chain, and 0 means the opposite. This representation is not suitable for large numbers and the authors experiment

with values of only up to 250. We note that the chromosome is of length 250 for that value, and for any value of practical interest the chromosome would amount to more than the memory of all computers in the world. The same authors focus on optimizing addition-subtraction chains with GAs [16]. They use the same representation and exponent values as in [15], which makes their work also far from applicable. They also experiment with addition-subtraction chains with a maximal value of 343 in [17].

Nedjah and de Macedo Mourelle use Ant Colony Optimization to find minimal addition chains working with exponent sizes of up to 128 bits [18]. However, since they do not provide the numbers themselves, but only their sizes, it is impossible to assess the quality of this approach besides the fact that they report it is better than the binary, quaternary, and octal method. The same authors extend their work for exponent sizes up to 1 024 bits resulting in better results for the Ant Colony Optimization algorithm than in cases when binary, quaternary, octal, and GA methods are used [19].

Cortés et al. propose a genetic algorithm approach for which the encoding is the chain itself [20]. Besides that, the authors also propose dedicated mutation and crossover operators. Using this approach, they report to successfully find minimal addition chains for numbers up to 14 143 037. Cortés, Rodríguez-Henríquez, and Coello present an Artificial Immune System for generating short addition chains of sizes up to 14 143 037 [21]. With that approach, the authors were successful in finding almost all optimal addition chains for exponents $e < 4\,096$.

Osorio et al. [22] propose a genetic algorithm coupled with a local search algorithm and repair mechanism in order to find minimal short addition chains. This work is of high relevance since it clearly discusses the need for a repair mechanism when using heuristics for the addition chains problem.

León-Javier et al. [23] experiment with the Particle Swarm Optimization algorithm in order to find optimal short addition chains. Nedjah and de Macedo Mourelle [24] implement the Ant Colony Optimization algorithm on a SoC in order to speed up the modular exponentiation in cryptographic applications. Sarkar and Mandal [25] use Particle Swarm Optimization to obtain faster modular multiplication in cryptographic applications for wireless communication.

Rodriguez-Cristerna and Torres-Jimenez [26] use a GA to find minimal Brauer chains where a Brauer chain is an addition chain in which each member uses the previous member as a summand. Finally, Domínguez-Isidro et al. [27, 28] investigate the usage of evolutionary programming for minimizing the length of addition chains.

4 The Design of the Proposed Algorithm

Before discussing the choice of the algorithm, we briefly enumerate some basic rules our chains need to fulfill:

1. Every chain (solution) needs to be an ascending chain.

2. Every chain needs to be non-redundant, i.e., there should not be two identical numbers in a chain.
3. Every chain needs to be valid, i.e., every number in a chain needs to be a sum of two previously appearing numbers.
4. Every chain needs to start with the value 1 and finish with the desired exponent value.

When choosing the appropriate algorithm for the evolution of chains, we start with the considerations about the representation. If we disregard the approach where one encodes individuals in a binary way (i.e., for each possible value, we use either zero if it is not a part of the chain, or one when it is a part of the chain), up to now there is not much of a choice. Indeed, encoding solutions as integer values where each value represents the number that occurs in the chain seems rather natural. Accordingly, we also use that representation, which we denote as encoding with *chain values*.

However, internally, our algorithm works with one more representation where we represent each value n as a pair of positions i_1 and i_2 that hold the previous values n_1 and n_2 forming the value n, which is denoted as encoding with *summand positions*.

Although such position based encoding gives longer chromosomes, for large exponents the encoded values are much smaller and the memory requirements for storing an individual are consequently smaller. Furthermore, it is possible to use operators that work on the positions and to give an algorithm more options to combine solutions (since we have two positions for every number, the length of a chain encoded with positions is always twice as long as the one encoded with values).

For both representations, a GA seems a natural choice, but there is one important difference in both approaches. When using the representation based on chain values for large numbers, the chromosome encoding needs to support large numbers, while in the representation based on summand positions we only need to support large numbers for calculating the chain elements, but not for storing them.

However, one cannot aim to fulfill the aforementioned rules and use a standard GA. Therefore, we need to design a custom initialization procedure, mutation, and crossover operators. In fact, only the selection algorithm can be used as in the standard GA. In all our experiments, we work with k-tournament selection where $k = 3$. In each tournament, the worst of k randomly selected individuals is replaced by the offspring of the best two from the same tournament.

Since initialization and variation operators are expected to produce many invalid solutions (in fact, for larger chains our experiments showed that it is highly unlikely that genetic operators will produce valid solutions) we also need to design a repair strategy. The repair strategy can be incorporated in each of the previous parts or to be considered as a special kind of operator, which is the approach we opted to follow. Next, we present the operators we use in our GA.

4.1 Initialization Algorithm

We design the initialization algorithm in a way to offer as much diversity as possible. We accomplish this by analyzing a number of known optimal chains (both star and standard chains) and checking the necessary steps to obtain them. Here, we note that if the initialization can produce only star chains and the mutation can generate only star steps, the whole algorithm will be able to produce only star chains. Naturally, one could circumvent this by adding additional steps in the repair mechanism. In that case, the model would not follow the intuition, since one expects that the repair mechanism only repairs the chains and it should not possess additional mechanisms for the generation of new values.

The initial population is generated via a set of hardcoded values that are positioned at the beginning of the chain and randomly generated chain sequences as presented below. The probability values are selected on the basis of a set of tuning experiments.

1. Set the zeroth element to 1 and the first element to 2.
2. Uniformly at random select between all minimal subchains consisting of three elements (i.e., the second, third, and fourth position in the chain) and a random choice of the second element (according to the rules, either the value 3 or 4).
3. With a probability equal to 3/5, double the elements until they reach half of the exponent size.
4. Check whether the current element and any previous element sum up to the exponent value.
5. Uniformly at random, choose among the following mechanisms to obtain the next value in the chain, under the constraint that it needs to be smaller than the exponent value:
   (a) Sum two preceding elements of the chain.
   (b) Sum the previous element and a random element.
   (c) Sum two random elements. One random element is chosen between the zeroth position and the element in the middle of the chain and the second one is chosen between the middle element and the final (exponent) value.
   (d) Loop from the element on the position $i - 1$ until the largest element that can be summed up with the last element is found.

4.2 Variation Operators

Next, we present the mutation and crossover operators we use. They are very similar to the operators provided, for instance, in [20, 21]. For such a specific problem as the one we study here, the task of devising new operators is difficult. Furthermore, many operators reduce to the ones described here. For instance, we present here something that is analogous to a single-point mutation, but since the change in a single position will invalidate the chain, after the repair mechanism, the mutation can also be regarded as a mixed mutation. Therefore,

the number of mutation points is irrelevant since a single point change brings changes in every position until the end of the chain.

Since we have several branches in the mutation operator, one can say that those branches could be separated into different mutation operators. We note that there are more possibilities on how to combine two values to form a new value in a sequence and there could be possibilities for additional mutation operators.

On the other hand, we implemented two crossover operators and we consider advantageous to use both of them, since this promotes diversity. However, identifying which of them is better than the other is hard, since it depends on the exponent value that we aim to reach.

Crossover. We implemented two versions on the crossover operator: one-point crossover and two-point crossover. We provide the pseudocode for one-point crossover in Algorithm 4.2 and note that the two-point version is analogous. Here, the function $FindLowestPair(P, i, pair_1, pair_2)$ determines the pair of elements with lowest indexes $(pair_1, pair_2)$ which give the target element i in a chain P. The dominant difference between the mutation operator and the crossover operator lies in the fact that in the crossover, we have defined the rules on how to build elements while in the mutation we do not have such strict rules. However, since both require the usage of the repair mechanism, that difference can become rather fuzzy.

Algorithm 1 Crossover operator.

Require: Exponent $exp > 0$, *Parent addition chains* P_1, P_2
  $rand = random(3, exp - 1)$
  for all i such that $0 \leq i \leq rand$ **do**
    $e_i = P_{1i}$
  end for
  for all i such that $rand \leq i + 1 \leq n$ **do**
    $FindLowestPair(P_2, i, pair_1, pair_2)$
    $e_i = e_{pair_1} + e_{pair_2}$
  end for
  $RepairChain(e, exp)$
  return $e = e_0, e_1, ..., e_n$

Mutation. The mutation operator is again similar to those presented in the related literature, but we allow more diversity in the generation process as presented in Algorithm 2. As already stated, since the mutation invalidates the chain, it is impossible to expect small changes (except when the mutation point is at the end of the chain) and therefore, this is actually a macromutation operator.

Algorithm 2 Mutation operator.

Require: Exponent $exp > 0, e = e_0, e_1, ..., e_n$
  $rand = random(2, exp - 1)$
  $rand_2 = random(0, 1)$
  if $rand_2$ **then**
    $e_{rand} = e_{rand-1} + e_{rand-2}$
  else
    $rand_3 = random(2, rand - 1)$
    $e_{rand} = e_{rand-1} + e_{rand_3}$
  end if
  $RepairChain(e, exp)$
  return $e = e_0, e_1, ..., e_n$

4.3 The Repair Algorithm

Function $RepairChain(e, exp)$ takes the chain e and repairs it in the following way:

1. Delete duplicate elements in the chain.
2. Delete elements greater than the *exp* value.
3. Check that all elements are in ascending order, if not, sort them.
4. Ensure that the chain finishes with the *exp* value by repeating operations in the following order:
   (a) Try to find two elements in the chain that result in *exp*.
   (b) Uniformly at random apply:
      i. Double the last element of the chain while it is smaller than *exp*.
      ii. Add the last element and a random element.
      iii. Add two random elements.

This function is in many ways similar to the Initialization procedure, but here with the primary goal of removing redundant chain elements, rather than maximizing diversity as is the case in the Initialization.

There are several places in our algorithm where we choose what branch to enter based on random values. We decided to use uniform random values where each branch has the same probability to be chosen. We believe this mechanism can be further improved. One trivial modification would be with regards to whether one wants to obtain a star chain or not. In the case when only star chains are wanted, then the branches that cannot result in a star step can be set either to a zero or some small value, analogous for the case when we want to have a larger number of standard steps.

4.4 The Fitness Function

We use a simple fitness function where the goal is *minimization*. The number of elements in the chain is minimized as given by the equation:

$$fitness = l(n). \tag{6}$$

5 Results and Discussion

5.1 Experimental Setup

The number of independent runs for each experiment is 50. For the stopping criterion we use stagnation which we set to 100 generations without improvement. We set the total number of generations to 1 500. The population size is set to 300 in all experiments. We note that larger population sizes perform even better thanks to increased diversity from the initialization mechanism, but for large exponent values the evolution then takes a long time. With the current setting, even for larger exponent size, one evolutionary run finishes in less than one hour.

5.2 Results

When discussing the efficiency of our algorithm, we need to establish a number of test cases that will:

1. serve as a comparison with previous work,
2. serve as special test cases and
3. serve as real-world benchmark tests.

Tests based on a comparison with previous work
For the first category, we used a set of exponent values that are also used in previous work. Namely, those are the exponents belonging to the class that is the most difficult to calculate according to [2]. Recall, those values are the minimal integers that form an addition chain of a certain length i. Up to now, experiments were done for values of i up to 30 [20, 21]. However, we wanted to evaluate the performance of our algorithm with even higher values and, therefore, we experimented with values up to $i = 40$. Furthermore, for each of those values we give statistical indicators in order to understand better the performance of our algorithm as well as to serve as a reference for future work.

Any comparison with previous work is difficult since it only reports the value (and the chain) that presents the best obtained solution. From the reproducibility and the efficiency side, we find that approach somewhat incomplete since it makes a difference if the algorithm found the best possible value in one instance out of 100 runs or in 90 instances out of 100 runs.

We note that for exponent values $n < 2^{27}$ one can find optimal chains online [6], while values of up to $n = 2^{31}$ can be obtained from the same web page. Therefore, in a sense, we conclude it is easy to compare with all values up to 2^{31} and we do not investigate such cases any further. However, as n increases, the situation changes since it becomes difficult to find any results for a direct comparison. Therefore, besides our algorithm, we implement the binary algorithm as well as two variants of the window method. In the first m-window method, we set the value of k to four in the expression $m = 2^k$. It has been shown [5] that with this method the length of the chain is:

$$l(n) \leq log_2(n) + 2^{k-1} - (k-1) + [log_2(n)/k], \forall k. \tag{7}$$

The second version of the window method tries to optimize Eq. (7) by choosing the value k that minimizes $2^{k-1} - (k-1) + [log_2(n)/k]$. We emphasize that none of the aforementioned methods should be regarded as the state-of-the-art, but as methods that give good results and should serve as the baseline cases. For smaller values of the exponent, the first window method gives far worse results than even the binary method and therefore we do not present such solutions. We omit the results for the first five values of r where the exponent value $c(r)$ is smaller than 10 since it is trivial to find optimal values in this case (recall Section 1 where we stated that the value 15 is the first exponent where the binary method is not the optimal choice). Additionally, the initialization part of our algorithm has all optimal combinations for the first five exponents hardcoded and therefore the comparison is not fair. The results are given in Table 1 where it is easy to observe that the GA performs better than the binary and optimized window methods.

Table 1. c(r) family of the exponent values.

r	c(r)	Binary	Optimized window	GA		
				Min	Avg	Stdev
5	11	5	5	5	5	0
6	19	6	11	6	6	0
7	29	7	11	7	7	0
8	47	9	12	8	8	0
9	71	9	13	9	9	0
10	127	12	13	10	10	0
11	191	13	14	11	11	0
12	379	14	16	12	12	0
13	607	15	17	13	13	0
14	1 087	16	18	14	14	0
15	1 903	18	18	15	15	0
16	3 583	21	19	16	16	0
17	6 271	20	20	17	17	0
18	11 231	23	22	18	18	0
19	18 287	23	23	19	19	0
20	34 303	25	24	20	20	0
21	65 131	26	24	21	21	0
22	110 591	30	25	22	22.08	0.27
23	196 591	32	27	23	23.04	0.19
24	357 887	32	28	24	24.28	1.26
25	685 951	33	29	25	25.1	0.58
26	1 176 431	33	31	26	26.18	1.27
27	2 211 837	36	32	27	27.18	1.68
28	4 169 527	37	32	28	28.18	0.38
29	7 624 319	36	33	29	30.16	0.71
30	14 143 037	38	34	30	30.92	0.6
31	25 450 463	38	35	31	32.62	0.66
32	46 444 543	42	36	32	33.5	0.54
33	89 209 343	42	38	33	34.46	0.81
34	155 691 199	42	39	34	35.44	1.03
35	298 695 487	46	41	35	35.67	0.74
36	550 040 063	45	41	36	37.96	0.83
37	994 660 991	46	42	37	38.76	1.47
38	1 886 023 151	48	42	38	40.28	1.21
39	3 502 562 143	48	43	39	41.36	1.19
40	6 490 123 999	52	45	41	41.77	0.63

Special test cases

Tests constituting special cases deal with the theoretical results we enumerated in Section 2. Here, we test the smallest value where $l(n) = l(2n)$ which is 191. Next, we test the smallest number n where the optimal chain is not a star chain, which is 12 509. We present the values that form the chain since it is interesting to observe several things. The smallest chain is $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 13 \rightarrow 24 \rightarrow 48 \rightarrow 96 \rightarrow 192 \rightarrow 384 \rightarrow 768 \rightarrow 781 \rightarrow 1\,562 \rightarrow 3\,124 \rightarrow 6\,248 \rightarrow 12\,496 \rightarrow 12\,509$. Note that this is not the only combination giving this chain of shortest length, but the following observations hold for others. Here, we are interested in values $12 \rightarrow 13 \rightarrow 24$, which is the part that does not follow the rules of a star chain.

If we compare this sequence with those obtainable from related work (cf. [20, 21]), we notice that in those approaches there exist no steps that can produce such a sequence. Therefore, although related work presented heuristic algorithms that are good on selected test cases, we show that they would not work for this case and therefore are not general enough for every addition chain, but only for star chains. The final special test case is the number $n = 375\,494\,703$ since $l(n) = 35$ while $l(2n) = 34$. Results for all special cases are given in Table 2. As in the first set of experiments, the GA approach again easily outperforms the binary and optimized window methods.

Table 2. Special test cases.

n	l(n)	Binary	Optimized window	GA		
				Min	Avg	Stdev
191	11	13	14	11	11	0
382	11	14	16	11	11.1	0.3
12 509	17	20	21	17	17.96	0.19
375 494 703	35	41	40	35	36.36	0.87
750 989 406	34	42	40	34	36.56	0.81

Real-world benchmark tests

As a real-world benchmark, we investigate values up to $2^{127} - 3$. We select that upper limit since it has applications in certain high speed Diffie-Hellman implementations [29]. To provide additional experiments for a comparison, we start with a value $2^{37} - 3$ and we progress by increasing the exponent in steps of ten, i.e., the following value is $2^{47} - 3$. We finish the experiments with the exponent $2^{127} - 3$ (170 141 183 460 469 231 731 687 303 715 884 105 725). We also present the results for the window method with a fixed value of k ($k = 4$) since it produces better results than the binary method. The results are given in Table 3. Similarly as in the previous cases, the GA approach is again superior while the differences between the results are even more striking than before.

We note that the shortest known chain for the exponent value $2^{127} - 3$ has 136 elements, which is the same value our algorithm reached. The question is whether this should be regarded as a success or a failure. In a sense, it depends on the perspective; if one knows that the value 136 was obtained (somewhat sur-

Table 3. Exponents up to $2^{127} - 3$.

Exponent	$log_2(n)$	$\nu(n)$	Binary	Window	Optimized win.	GA		
						Min	Avg	Stdev
$2^{37} - 3$	36	35	71	57	51	43	45.32	0.99
$2^{47} - 3$	46	45	91	69	63	54	56.25	1.11
$2^{57} - 3$	56	55	111	82	76	64	64.9	0.87
$2^{67} - 3$	66	65	131	94	88	73	73.2	0.43
$2^{77} - 3$	76	75	151	107	101	85	85.4	0.51
$2^{87} - 3$	86	85	171	119	113	97	104.3	3.56
$2^{97} - 3$	96	95	191	132	126	106	107.2	0.91
$2^{107} - 3$	106	105	211	144	138	115	115.71	0.75
$2^{117} - 3$	116	115	231	157	151	126	126.6	0.89
$2^{127} - 3$	126	125	251	169	163	136	136.8	0.83

prising) by a pen and paper approach in a matter of a few hours by an expert, then our result does not seem impressive. However, recall Definition 1 where it is easy to calculate that $l(2^{127} - 3)$ has a chain of a length at least equal to 130 since this exponent has 125 ones in its binary representation. On the other hand, the GA found the chain of the same length without any problems and in less than 30 minutes on average. Furthermore, maybe there are no shorter chains for that exponent, so the GA actually reaches the optimal value. Unfortunately, the answer to this question seems out of our reach without some new analytical breakthrough or until the processing power increases sufficiently to run an exhaustive search. Since both of those perspectives are unlikely at this moment, we consider our algorithm useful since it gives us an option to effortlessly find many short chains for a wide range of exponent values.

6 Conclusions and Future Work

In this work, we showed that GAs can be used to find shortest addition chains for a wide set of exponent sizes. However, we note this problem is not as easy as could be perceived from a number of related works. Indeed, the first step is the design of a custom GA and then one needs to carefully tune the parameters. Here, we managed to find chains that are either optimal (where it was possible to confirm based on related work) or as short as possible for a number of values. From that perspective, we see this work also as a reference work against which new heuristics should be tested, since it is undoubtedly possible to compare the results. As far as we know, we are the first to investigate this kind of heuristics for an exponent value that has a real world usage.

As part of our future work we plan to investigate even larger values that are useful in practice. We also note that our position based representation actually corresponds to the Cartesian Genetic Programming (CGP) encoding. There, we always use one function (plus) and for each node the indexes from the two previous nodes are recorded, which can be encoded as a graph of size $1 \times N$, which motivates us to experiment with CGP in the future.

7 Acknowledgments

This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882. The second author gratefully acknowledges support from CONACyT project no. 221551. In addition, this work was supported in part by the Research Council KU Leuven (C16/15/058) and IOF project EDA-DSE (HB/13/020).

References

1. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2) (1978) 120–126
2. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing, Boston, MA, USA (1997)
3. Menezes, A., van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. CRC Press, Boca Raton, Florida, USA (1996)
4. Thurber, E.G.: The scholz-brauer problem on addition chains. Pacific Journal of Mathematics **49**(1) (1973) 229–242
5. Thurber, E.G.: On addition chains $1(mn) \leq 1(n) - b$ and lower bounds for $c(r)$. Duke Mathematical Journal **40**(4) (12 1973) 907–913
6. Flammenkamp, A.: Shortest Addition Chains (nov 2015) http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html.
7. Bernstein, D.J.: Differential addition chains (2006) https://cr.yp.to/ecdh/diffchain-20060219.pdf.
8. Gordon, D.M.: A Survey of Fast Exponentiation Methods. Journal of Algorithms **27** (1998) 129–146
9. Gallant, R., Lambert, R., Vanstone, S.: Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In Kilian, J., ed.: Advances in Cryptology CRYPTO 2001. Volume 2139 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 190–200
10. Galbraith, S., Lin, X., Scott, M.: Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. Journal of Cryptology **24**(3) (2011) 446–469
11. Costello, C., Longa, P.: FourQ: four-dimensional decompositions on a Q-curve over the Mersenne prime. Cryptology ePrint Archive, Report 2015/565 (2015) http://eprint.iacr.org/.
12. Faz-Hernández, A., Longa, P., Sánchez, A.: Efficient and Secure Algorithms for GLV-Based Scalar Multiplication and Their Implementation on GLV-GLS Curves. In Benaloh, J., ed.: Topics in Cryptology CT-RSA 2014. Volume 8366 of Lecture Notes in Computer Science. Springer International Publishing (2014) 1–27
13. Coron, J.S.: Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems. In Ko, e., Paar, C., eds.: Cryptographic Hardware and Embedded Systems. Volume 1717 of Lect. Notes in Comp. Science. Springer (1999) 292–302
14. Bos, J., Coster, M.: Addition Chain Heuristics. In Brassard, G., ed.: Advances in Cryptology - CRYPTO'89 Proceedings. Volume 435 of Lecture Notes in Computer Science. Springer New York (1990) 400–407
15. Nedjah, N., de Macedo Mourelle, L.: Minimal Addition Chain for Efficient Modular Exponentiation Using Genetic Algorithms. In Hendtlass, T., Ali, M., eds.: Developments in Applied Artificial Intelligence. Volume 2358 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2002) 88–98

16. Nedjah, N., de Macedo Mourelle, L.: Minimal Addition-Subtraction Chains Using Genetic Algorithms. In: Advances in Information Systems. Volume 2457 of Lect. Notes in Comp. Science. Springer (2002) 303–313
17. Nedjah, N., de Macedo Mourelle, L.: Minimal Addition-Subtraction Sequences for Efficient Pre-processing in Large Window-Based Modular Exponentiation Using Genetic Algorithms. In Liu, J., Cheung, Y.m., Yin, H., eds.: Intelligent Data Engineering and Automated Learning. Volume 2690 of Lect. Notes in Comp. Science. Springer (2003) 329–336
18. Nedjah, N., de Macedo Mourelle, L.: Finding Minimal Addition Chains Using Ant Colony. In Yang, Z., Yin, H., Everson, R., eds.: Intelligent Data Engineering and Automated Learning - IDEAL 2004. Volume 3177 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 642–647
19. Nedjah, N., de Macedo Mourelle, L.: Towards Minimal Addition Chains Using Ant Colony Optimisation. Journal of Mathematical Modelling and Algorithms **5**(4) (2006) 525–543
20. Cruz-Cortés, N., Rodrguez-Henrquez, F., Juárez-Morales, R., Coello Coello, C.: Finding Optimal Addition Chains Using a Genetic Algorithm Approach. In Hao, Y., Liu, J., Wang, Y., Cheung, Y.m., Yin, H., Jiao, L., Ma, J., Jiao, Y.C., eds.: Computational Intelligence and Security. Volume 3801 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 208–215
21. Cruz-Corteés, N., Rodriguez-Henriquez, F., Coello Coello, C.: An Artificial Immune System Heuristic for Generating Short Addition Chains. Evolutionary Computation, IEEE Transactions on **12**(1) (Feb 2008) 1–24
22. Osorio-Hernández, L.G., Mezura-Montes, E., Cortés, N.C., Rodríguez-Henríquez, F.: A genetic algorithm with repair and local search mechanisms able to find minimal length addition chains for small exponents. In: Proc. IEEE Congress on Evolutionary Computation, Trondheim, Norway, 18-21 May. (2009) 1422–1429
23. León-Javier, A., Cruz-Cortés, N., Moreno-Armendáriz, M., Orantes-Jiménez, S.: Finding Minimal Addition Chains with a Particle Swarm Optimization Algorithm. In: MICAI 2009: Advances in Artificial Intelligence. Volume 5845 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2009) 680–691
24. Nedjah, N., de Macedo Mourelle, L.: High-performance SoC-based Implementation of Modular Exponentiation Using Evolutionary Addition Chains for Efficient Cryptography. Applied Soft Computing **11**(7) (October 2011) 4302–4311
25. Sarkar, A., Mandal, J.: Swarm Intelligence based Faster Public-Key Cryptography in Wireless Communication (SIFPKC). International Journal of Computer Science & Engineering Technology (IJCSET) (7) (2012) 267–273
26. Rodriguez-Cristerna, A., Torres-Jimenez, J.: A Genetic Algorithm for the Problem of Minimal Brauer Chains. In: Recent Advances on Hybrid Intelligent Systems. Volume 451 of Studies in Comp. Int. Springer Berlin Heidelberg (2013) 481–500
27. Domínguez-Isidro, S., Mezura-Montes, E., Osorio-Hernández, L.G.: Addition chain length minimization with evolutionary programming. In: 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011. (2011) 59–60
28. Domínguez-Isidro, S., Mezura-Montes, E., Osorio-Hernández, L.G.: Evolutionary programming for the length minimization of addition chains. Eng. Appl. of AI **37** (2015) 125–134
29. Bernstein, D.J., Chuengsatiansup, C., Lange, T., Schwabe, P.: Kummer strikes back: new DH speed records. In Iwata, T., Sarkar, P., eds.: Advances in Cryptology – EUROCRYPT 2015. Volume 8873 of Lecture Notes in Computer Science., Springer-Verlag Berlin Heidelberg (2014) 317–337